Павел Филонов

Угадайте язык по слогану

- Напишите один раз, запускайте где угодно
  - Java
- Не все, что делает Microsoft получается плохо
  - C#
- Красота лучше уродства
  - Python
- C++

Bjarne Stroustrup: C++ Zero-Overhead Principle and OOP

Zero-overhead principle

1. You don’t pay for what you don’t use
2. What you do use is just as efficient as what you could reasonably write by hand

Почему это важно?

The benchmark game

How many times more CPU seconds?

program cpu time / lowest

300
100
50
30
10
5
3
1

C gcc
C++ g++
Rust
C# .NET
Classic Fortran
Julia
Chapel
Ada 2012 GNAT
Haskell GHC
OCaml
Go
Swift
F# .NET
Java
Free Pascal

benchmarks game

10 May 2022 u64q

Как показать данный принцип в действии?

Rust Dublin

Zero-Cost Abstractions

Michael Barber, April 2021

With thanks to our sponsors

Тестовая задача

```
sum = 0
for every pair a, b in aligned vectors va, vb
    if a > 2
        sum += a * b
```

Условия теста производительности:

va и vb размером в 20k элементов, которые заданы целыми числами в полуинтервале [0, 10).
Для замеров случайно выбираем va и vb из 100 заранее сгенерированных векторов.

Тестовый стенд:

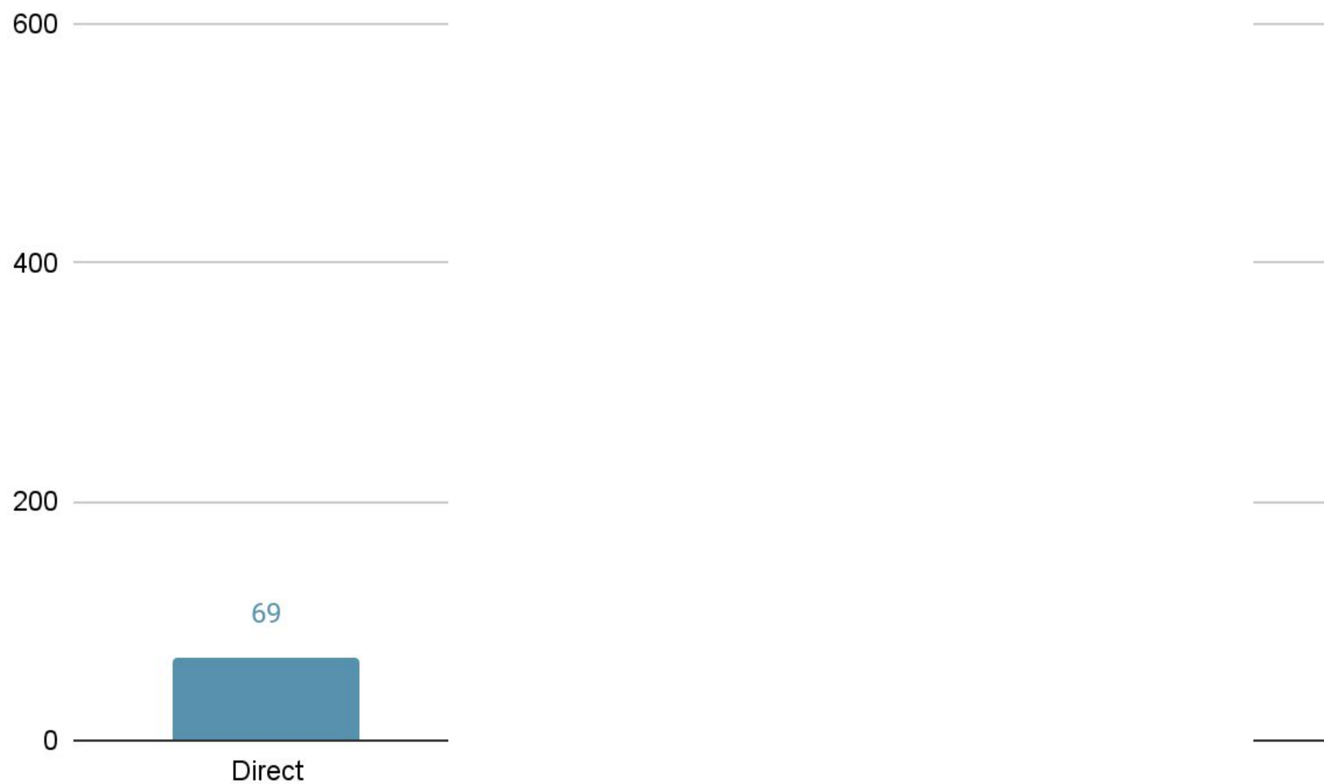
Intel Xeon (Icelake) 2Ghz, Ubuntu 20.04, все тесты в 1 поток
.NET Core 6.0.5, Clang++ 14.0.1, Rustc 1.60

C#

C# прямой цикл

```
public static long CalculateDirect(int[] va, int[] vb) {
    long sum = 0;
    for (var i = 0; i < va.Length; ++i) {
        var a = va[i];
        var b = vb[i];
        if (a > 2) {
            sum += a * b;
        }
    }
    return sum;
}
```

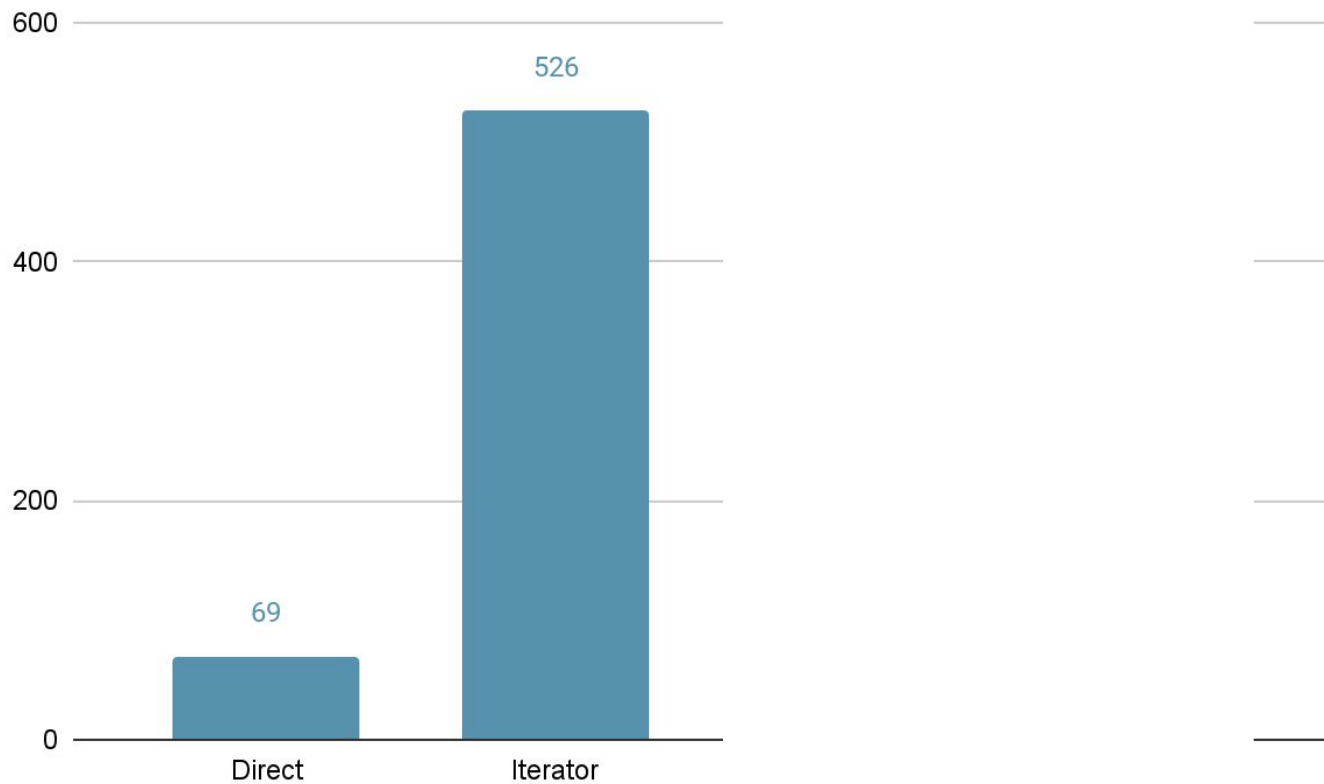
C# результаты



C# итератор

```
public static long CalculateIterator(int[] va, int[] vb)
{
    var res = va.Zip(vb)
        .Where(pair => pair.First > 2)
        .Select(pair => (long)(pair.First * pair.Second))
        .Sum();
    return res;
}
```

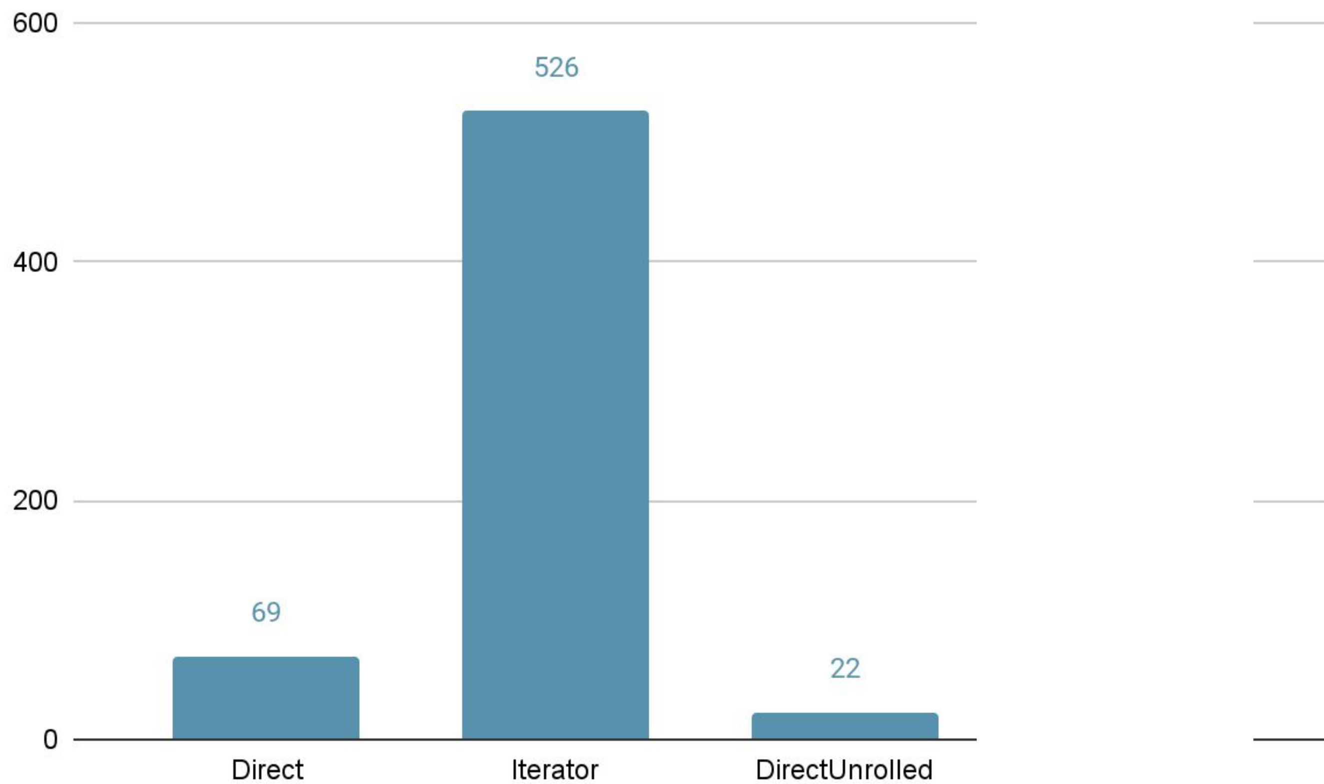
C# результаты



C# раскрученный цикл

```
public static long CalculateDirectUnrolled(int[] va, int[] vb)
{
    long sum = 0;

    var len = va.Length;
    var spana = va.AsSpan(0, len);
    var spanb = vb.AsSpan(0, len);
    var chunkEnd = (len >> 2) << 2;

    var i = 0;
    while (i < chunkEnd) {
        var sa = spana.Slice(i, 4);
        var sb = spanb.Slice(i, 4);

        var m0 = MaskGreaterThan(sa[0], 2);
        var m1 = MaskGreaterThan(sa[1], 2);
        var m2 = MaskGreaterThan(sa[2], 2);
        var m3 = MaskGreaterThan(sa[3], 2);

        var v0 = m0 & (sa[0] * sb[0]);
        var v1 = m1 & (sa[1] * sb[1]);
        var v2 = m2 & (sa[2] * sb[2]);
        var v3 = m3 & (sa[3] * sb[3]);

        var v01 = v0 + v1;
        var v23 = v2 + v3;
        sum += v01;
        sum += v23;

        i += 4;
    }
    // eval rest of data
    return sum;
}
```

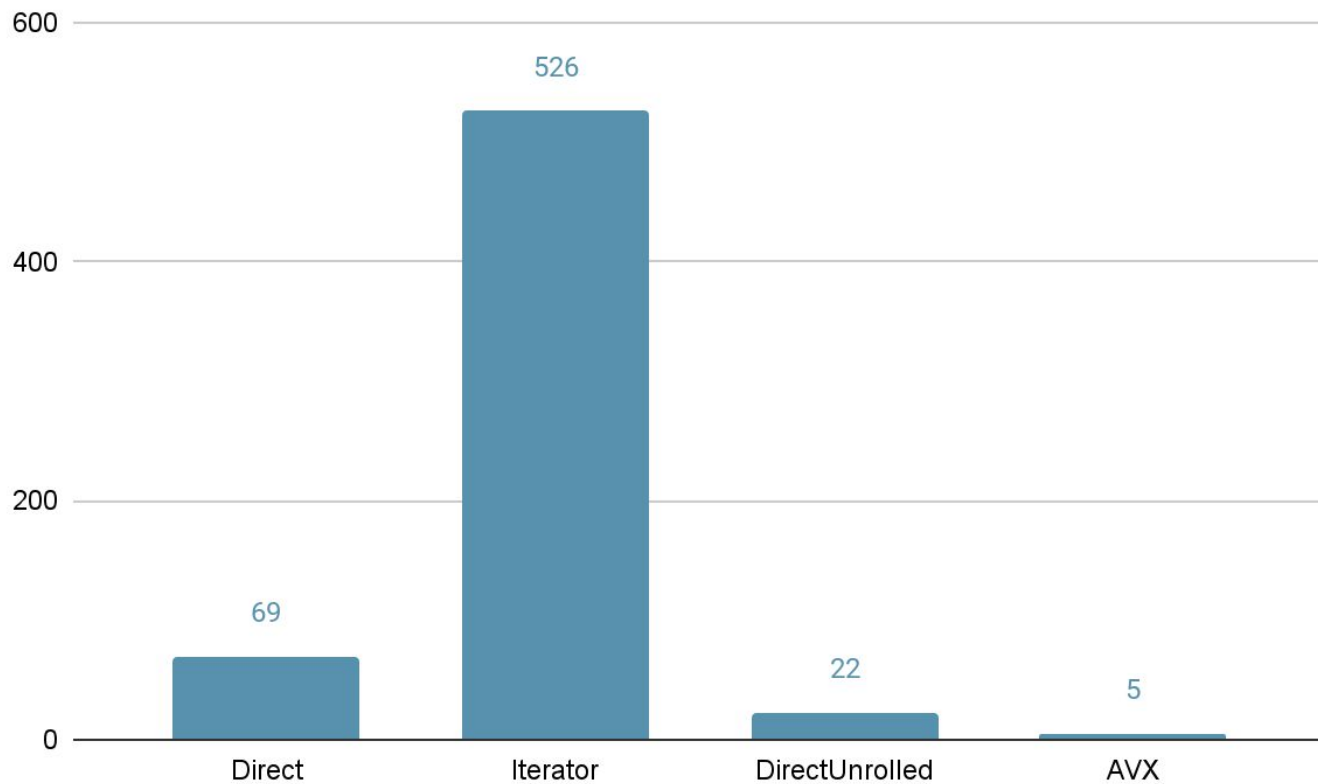
C# результаты



C# AVX

```
public static long CalculateDirectUnsafeAvx(int[] vecta, int[] vectb)
{
    if (vecta.Length != vectb.Length) throw new ArgumentException("length mismatch");

    long sum = 0;
    var len = vecta.Length;
    var chunkEndIndex = (len >> 3) << 3;
    unsafe {
        fixed (int* ptra = vecta, ptrb = vectb) {
            var chunkEnd = ptra + chunkEndIndex;
            var allEnd = ptra + len;
            var pa = ptra;
            var pb = ptrb;

            var value2 = Vector256.Create(2);
            var value0 = Vector256.Create(0);

            // two accumulators, each 4 x int64, for total 8 x int64
            var acc1 = Vector256.Create(0L);
            var acc2 = Vector256.Create(0L);

            // main loop -- 8 elements at a time
            while (pa < chunkEnd)
            {
                var a = Avx.LoadVector256(pa);
                var b = Avx.LoadVector256(pb);

                var mask = Avx2.CompareGreaterThan(a, value2);
                a = Avx2.And(a, mask);

                // odd numbered elements (i32 * i32 -> i64)
                var m1 = Avx2.Multiply(a, b);
                acc1 = Avx2.Add(acc1, m1);

                // shuffle adjacent and multiply again
                a = Avx2.Shuffle(a, 0b10_11_00_01);
                b = Avx2.Shuffle(b, 0b10_11_00_01);
                var m2 = Avx2.Multiply(a, b);
                acc2 = Avx2.Add(acc2, m2);

                pa += 8;
                pb += 8;
            }
            // tail loop -- single elements
            while (pa < allEnd)
            {
                // could do something smart like a masked load here, but
                // this will do for demonstration purposes.
                var a = Vector256.CreateScalar(*pa);
                var b = Vector256.CreateScalar(*pb);

                // as above, accumulating into acc1
                var mask = Avx2.CompareGreaterThan(a, value2);
                a = Avx2.And(a, mask);
                var m1 = Avx2.Multiply(a, b);
                acc1 = Avx2.Add(acc1, m1);

                pa++;
                pb++;
            }

            // sum all the elements
            acc1 = Avx2.Add(acc1, acc2);
            sum = acc1.GetElement(0) + acc1.GetElement(1) +
                  acc1.GetElement(2) + acc1.GetElement(3);
        }
    }
    return sum;
}
```

C# результаты

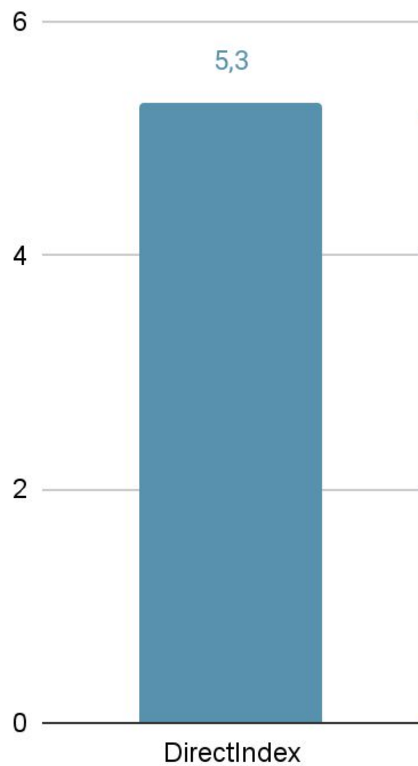


C++

C++ простой цикл

```
std::int64_t calculate_direct_index(const std::vector<std::int32_t> &a,
                                    const std::vector<std::int32_t> &b)
{
    assert(a.size() == b.size());
    const size_t len = a.size();
    std::int64_t res = 0;
    for (size_t i = 0; i < len; ++i)
    {
        if (a[i] > 2)
        {
            res += a[i] * b[i];
        }
    }
    return res;
}
```

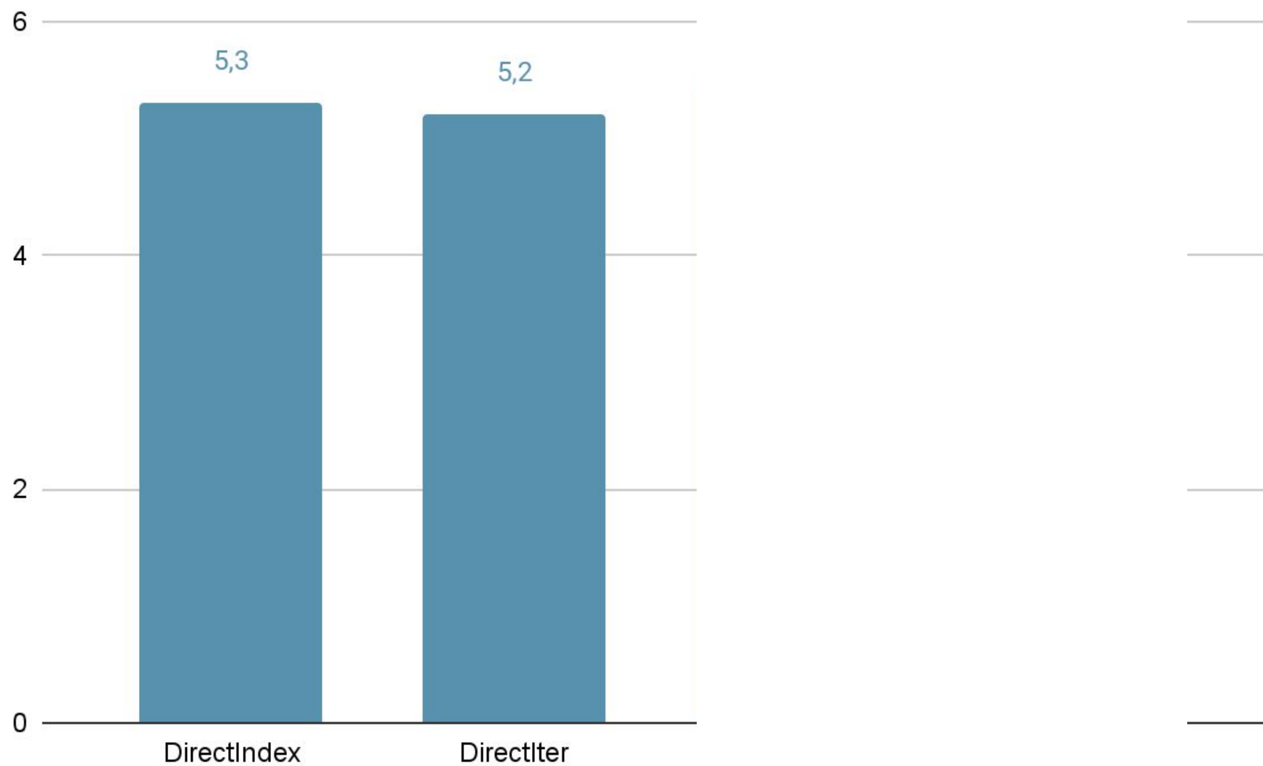
C++ результаты



C++ прямой цикл с итератором

```
std::int64_t calculate_direct(const std::vector<std::int32_t> &a,
                              const std::vector<std::int32_t> &b)
{
    assert(a.size() == b.size());
    std::int64_t res = 0;

    for (const auto& [a, b] : zip(a, b)) {
        if (a > 2) {
            res += a * b;
        }
    }

    return res;
}
```

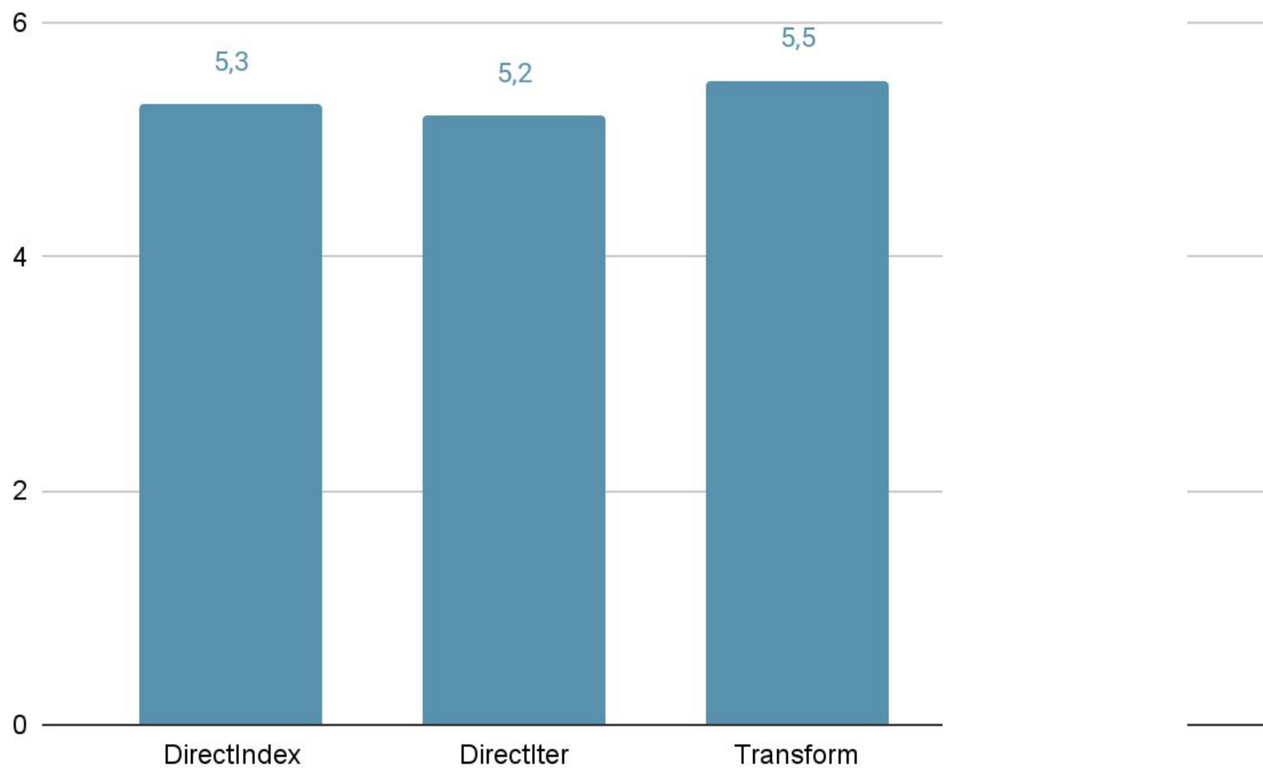
C++ результаты



C++ transform_reduce

```
std::int64_t calculate_tranform_reduce(const std::vector<std::int32_t> &a,
                                       const std::vector<std::int32_t> &b)
{

    assert(a.size() == b.size());
    std::int64_t res = std::transform_reduce(
                             a.begin(), a.end(), b.begin(), 0,
                             std::plus<>(),
                             [](std::int32_t a, std::int32_t b)
                             { return (a > 2) ? a * b : 0; });

    return res;
}
```

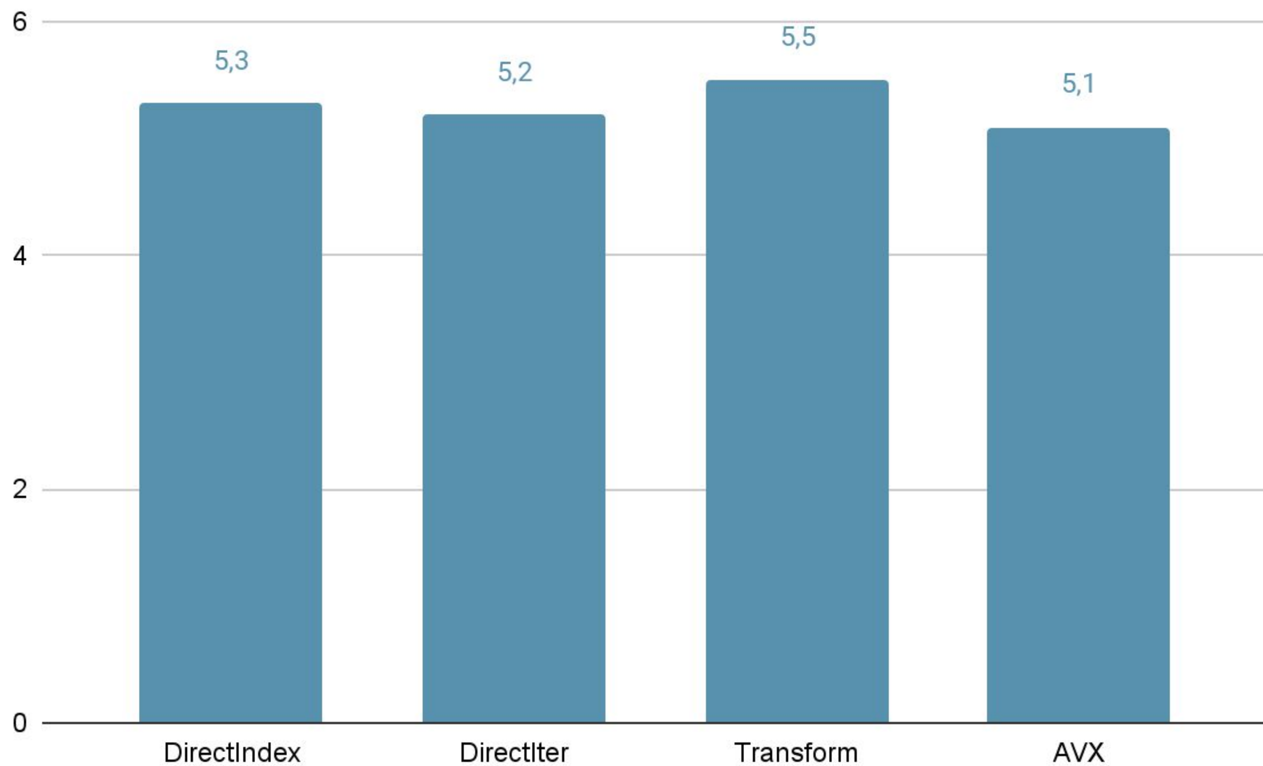
C++ результаты



C++ AVX

```cpp
std::int64_t calculate_avx(const std::vector<std::int32_t> &a,
                           const std::vector<std::int32_t> &b)
{
    assert(a.size() == b.size());
    const size_t size = 8; // initial size of chunks
    const size_t vec_length = a.size();

    auto value2 = _mm256_set1_epi32(2); // broadcast 2
    auto acc1 = _mm256_setzero_si256();
    auto acc2 = _mm256_setzero_si256();
    size_t i = 0;
    for (; i + size < vec_length; i += size) {
        auto va = _mm256_loadu_si256(reinterpret_cast<const __m256i_u*>(&a[i]));
        auto vb = _mm256_loadu_si256(reinterpret_cast<const __m256i_u*>(&b[i]));

        auto mask = _mm256_cmpgt_epi32(va, value2);
        va = _mm256_and_si256(va, mask);

        auto m1 = _mm256_mul_epi32(va, vb);
        acc1 = _mm256_add_epi64(acc1, m1);

        auto shuf_va = _mm256_shuffle_epi32(va, 0b10110001);
        auto shuf_vb = _mm256_shuffle_epi32(vb, 0b10110001);
        auto m2 = _mm256_mul_epi32(shuf_va, shuf_vb);
        acc2 = _mm256_add_epi64(acc2, m2);
    }

    std::int64_t remainder = 0;
    for (; i < vec_length; ++i) {
        if (a[i] > 2) {
            remainder += a[i] * b[i];
        }
    }

    auto acc = _mm256_add_epi64(acc1, acc2);
    std::int64_t d[size / 2];
    _mm256_storeu_si256(reinterpret_cast<__m256i_u *>(d),
acc);

    return std::accumulate(std::begin(d), std::end(d),
remainder);
}
```

C++ результаты

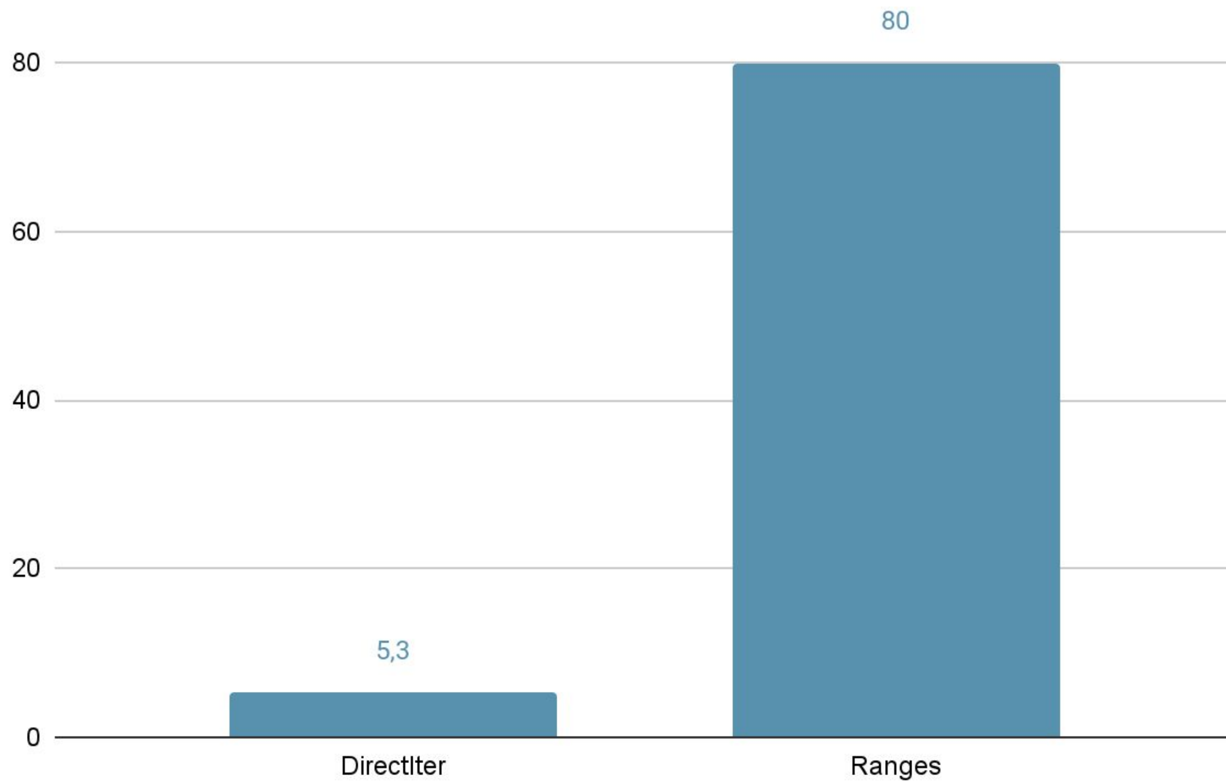


Всегда ли в C++ соблюдается принцип **zero-overhead?**

C++ ranges

```
std::int64_t calculate_ranges(const std::vector<std::int32_t> &a,
                              const std::vector<std::int32_t> &b)
{
    assert(a.size() == b.size());
    std::int64_t res = ranges::accumulate(ranges::views::zip(a, b)
        | ranges::views::filter([](auto &&values) { const auto& [a, b] = values; return a > 2; })
        | ranges::views::transform([](auto &&values) { const auto& [a, b] = values; return a * b; })
        , 0);

    return res;
}
```

C++ результаты



Источники

- The Benchmarks Game
- Rust: Zero Cost Abstractions
- zero-cost-abstractions benchmarks fork
- Intel Intrinsics Guide
- C++ ranges library
- Zero overhead principle

Спасибо за внимание!

@filonovpv
filonovpv@gmail.com

Rust результаты